Comparing different approachs for matrix multiplication omptimization

Eduardo López Fortes

January 12, 2025

Github repo: Fourth-MatrixMultiplication

Abstract

Through this memory we will discuss the use of hazelcast, not just for matrix multiplication but also as a solution for deploying a cluster, allowing developers to use the power of distributed computing. Regarding hazelcast, the different data structures that can be used will also be discussed, as well as how they can be used to solve the matrix multiplication problem in a distributed way. We will use Java as the programming language to implement the solution. As an additional problem it will be shown how to solve the frequent item set problem in a distributed way but using another distributed framework for distributed computing, MapReduce. MapReduce is a programming model that allows to process large data sets in a distributed way. The problem of frequent item set is a problem that is used in data mining to find the most frequent items in a data set. The solution will be implemented in Python using the mrjob library. In previous stages the implemented solutions would be benchmarked and compared, but due to the complexity of the problem that will not be possible. On the one hand it would be very difficult to create a cluster of computers since mine is the unique one available and therefore it would have no sense to benchmark it since it will not be a distributed solution. On the other hand, benchmarking the solution of the frequent item set problem would be useless since there is no other solution to compare it with. Therefore, the performance of the solutions will be discussed theoretically.

1 Introduction

This is the last stage of the project, so this stage will not be focused only on how to implement the solution in a distributed way, but it will also be a recap of the whole project. In previous it has been deeply discussed the matrix multiplication problem and multiple type of solution have been addressed: in the first stage it was discussed the naive solution, along with the question of which was the best programming lanaguage to perform the solution. In the second stage, rather than discussing which was the best environment to implement the solution, it was discussed how to improve the algorithm of matrix multiplication itself, in order to make it faster and more efficient: algorithms such as the Strassen method, the divide and conquer method, or the sparse matrix method were discussed among others were implemented, benchmarked and compared. In the third stage, the task was mainly focused on the divide and conquer method, but in a parallel way, instead of a sequential way: threads and streams among others were used to implement the solution. In this and last stage, the focus will be on how to implement the solution in a distributed way, using hazelcast and how to solve the frequent item set problem in a distributed way, but using another distributed framework for distributed computing, MapReduce.

Distributing computing is a way of computing that allows to process large data sets in a distributed way, using multiple computers. This is a way of computing that is used when the data set is too large to be processed in a single computer. In this way, the data set is divided into smaller data sets and each of them is processed in a different computer. This is a way of computing that allows to process large data sets in a faster way, since the data set is divided into smaller data sets and each of them is processed in a different computer. This way we can take advantage of the power of multiple computers to process the data set in a faster way. However, those computers need to be connected to each other, so they can communicate and share the data. This is where hazelcast comes into play. Hazelcast is a distributed computing platform that allows to deploy a cluster of computers, so they can communicate and share the data set. Hazelcast provides a set of data structures that can be used to store the data set and process it in a distributed way. In this way, hazelcast allows to process large data sets in a distributed way, using multiple computers.

As it is has been proved in the group assignment, distributed computing can be used for a wide range of problems, not only for matrix multiplication. There are different approaches on how to implement a solution in a distributed way: on one hand, we can have dedicated nodes, i.e. nodes that are responsible for a specific task. On the other hand, we can have generic nodes, i.e. nodes that are responsible for processing the data set in a generic way. In this way, we can have a set of nodes that are responsible for processing the data set in a generic way, and a set of nodes that are responsible for processing the data set in a specific way. In the case that concern us, the decision very clear and it will be discussed in the section of problem statement.

The matrix multiplication case study does not need any introduction, since it has been discussed in previous stages. However, the frequent item set problem does need an introduction. The frequent item set problem is a problem that is used in data mining to find the most frequent items in a data set. The problem is defined as follows: given a data set, find the most frequent items in the data set. The solution to this problem is to find the items that appear in the data set more than a given threshold. This approach is widely used in supermarkets to find the most frequent items that are bought together. If the supermarket knows which items are bought together, it can make better decisions on how to place the items in the store. If two products such as oil and milk are bought together, it makes sense to place them as far as possible in the store, so the customer has to walk through the whole store to buy them. This way the supermarket can increase the sales of other products.

The solution to this problem is to find the items that appear in the data set more than a given threshold. This is a problem that can be solved in a distributed way, using MapReduce. MapReduce is a programming model that allows to process large data sets in a distributed way. The model is based on two main functions: the map function and the reduce function. The map function is responsible for processing the data set and emitting key-value pairs. The reduce function is responsible for processing the key-value pairs and emitting the final result. In this way, MapReduce allows to process large data sets in a distributed way, using multiple computers.

2 Problem Statement

2.1 Matrix Multiplication

The problem that will be addressed in this stage is how to implement the matrix multiplication problem in a distributed way, using hazelcast. The problem is defined as follows: given two matrices A and B, find the product of the two matrices. The solution to this problem is to multiply each element of the first matrix by each element of the second matrix and sum the results. This is a problem that can be solved in a distributed way, using hazelcast. Hazelcast provides a set of data structures that can be used to store the matrices and process them in a distributed way. In this way, hazelcast allows to process large matrices in a distributed way, using multiple computers.

The first subproblem that needs to be addressed is how to divide the matrix into smaller matrices and how to represent them and how to divide the work among the different nodes. The easiest solution is to publish the job in a queue and let the nodes take the job from the queue. Then the nodes will process the job and publish the result in another queue. This way, the nodes can process the data set in a distributed way, using multiple computers. However, for this solution to work we need some node to send the jobs to the queue and then to collect the results from the queue. This brings us to the second subproblem that needs to be addressed, the nodes hierarchy. The nodes hierarchy is the way the nodes are organized in the cluster. Since the problem is somehow trivial, the nodes hierarchy will be very simple: thre will be a manager node that will be responsible for sending the jobs to the queue and collecting the results from the queue, and there will be worker nodes that will be responsible for processing the jobs. This way, the manager node will be responsible for coordinating the work of the worker nodes and the worker nodes will be responsible for processing the jobs. Unlike most of the managers in real life, this manager node, once it has sent all the jobs to the queue, it will also help the worker nodes to process the jobs. The main difference is that instead of publishing the results to the results queue, it will directly store the results in the matrix C. Once all the jobs have been processed and the jobs queue is empty, the manager node will be polling the multiplication results from the results queue and storing them in the matrix C.

Once the structure of the underlying system has been defined, the next step is to define the data structures that will be used to store the matrices and how to divide the matrices in smaller matrices. The best solution would be probably to split the two matrices in row * column multiplications. That way the job consists in multiplying a row of the first matrix by a column of the second matrix. This way, the job can be easily divided among the different nodes. The manager node will be responsible for dividing the matrices in smaller matrices and sending the jobs to the queue. Instead of using a Hazelcast data structure such as IMap, an artificial serializable data structure will be used to store the matrices. This data structure will contain all the necessary information for both the manager node and the worker nodes to process the jobs: the position of the resulting element in the matrix C, the row of the first matrix and the column of the second matrix that need to be multiplied, and a field to store the result of the multiplication. This way, the manager node can easily divide the matrices in smaller matrices and send the jobs to the queue, and the worker nodes can easily process the jobs and store the results in the same object before sending it to the results queue.

2.2 Frequent Item Set Problem

As exposed in the introduction, the frequent item set problem is a problem that is used in data mining to find the most frequent items in a data set. The problem is defined as follows: given a data set, find the most frequent items in the data set. The solution to this problem is to find the items that appear in the data set more than a given threshold. One of the best solutions to this problem is to use MapReduce. MapReduce is a programming model that allows to process large data sets in a distributed way. The model is based on two main functions: the map function and the reduce function. The map function is responsible for processing the data set and emitting key-value pairs. The reduce function is responsible for processing the key-value pairs and emitting the final result. In this way, MapReduce allows to process large data sets in a distributed way, using multiple computers.

Imagine the following scenario, one of the simplest ones: we want to count the appearances of each word in a text. The solution is simple: we can use the map function to split the text into words and emit a key-value pair for each word, where the key is the word and the value is 1. Then we can use the reduce function to count the appearances of each word and emit the final result. This is a simple example of how MapReduce can be used to solve the frequent item set problem.

The situation that will be addressed in this stage is how to implement the frequent item set problem in a distributed way, using MapReduce. The problem is defined as follows: given a list of baskets, find the most frequent items that are bought together. We could just simply count the appearances of each item in the baskets, but that would not be very useful. The solution to this problem is to find the items that are bought together, and we can achieve that by counting the appearances of each pair, triple, quadruple... of items in the baskets. Later in this memory it will be shown how to implement the solution in a distributed way, using MapReduce and the mrjob library.

3 Methodology

3.1 Matrix Multiplication

I am not a big fan of putting raw code into the memory (I consider that the use of pseudocode makes it easier to understand) but I consider that in this case it is necessary in order to show the reader how the cluster and the different data structures were designed and created.

3.1.1 Matrix Generation

Before implementing the algorithms, the method in which the matrices will be generated must be defined. The matrices will be generated randomly, and the elements will be double values between 0 and 1. The size of the matrices will be defined as a parameter, so that the algorithms can be tested with different matrix sizes. The matrices will be generated using the Random class from the java.util package. The matrices will be generated in the main method of the project, and then they will be passed as input to the algorithms. The pseudo-code of the matrix generation is as follows:

Algorithm 1 Matrix Generation

```
1: Input: $n$                                    ▷ Size of the matrices
2: Initialize matrices $a[n][n]$ and $b[n][n]$
3: Create random object $random$
4: for $i \leftarrow 0$ to $n-1$ do
5:     for $j \leftarrow 0$ to $n-1$ do
6:         $a[i][j] \leftarrow random.nextDouble()$
7:         $b[i][j] \leftarrow random.nextDouble()$
8:     end for
9: end for
```

3.1.2 Cluster Deployment

As told previously, the cluster will be deployed using hazelcast. Hazelcast is a distributed computing platform that allows to deploy a cluster of computers, so they can communicate and share different data structures. For this exmperiment, the cluster will be deployed in a single computer, but in real life the cluster can be deployed in several computers. The first task is to set up the cluster. In hazelcast there are two different options to do this (needless to say that all the computers in the cluster have to reside in the same network): the first way is to use the hazelcast.xml file, and the second way is to use the programmatic configuration. The first way is the easiest one, since it only requires to create a hazelcast.xml file and put it in the classpath. The second option is more complex, since it requires to create a configuration object and set the properties of the cluster. In this case, the second option will be used since I have some experience with it. There are two options for the different nodes to join the cluster:

- Using multicast configuration: this option is the easiest one and the most flexible one, since it allows the nodes to join the cluster without knowing the IP address of the other nodes. However, I have never make it work, so I will not use it.

- Using TCP/IP configuration: this option is the most secure one, since it requires the nodes to know the IP address of the other nodes. This way, only the nodes that know the IP address of the other nodes can join the cluster. This is the option that will be used. If the computers have several network interfaces, the IP address of the network interface that will be used to join the cluster has to be specified. Since all the computers have to be connected to the same network, all of the nodes have to use the same network interface. You can specify the IP address of each node by using the setPublicAddress method of the TcpIpConfig object. This way, the nodes can join the cluster using the IP address of the network interface that will be used to join the cluster.

```java
Config config = new Config();
config.getNetworkConfig().getJoin().getMulticastConfig().setEnabled(false);
config.getNetworkConfig().getJoin().getTcpIpConfig()
        .setEnabled(true)
        .addMember("10.26.14.210")
        .addMember("10.26.14.211")
        .addMember("10.26.14.212");
config.getNetworkConfig().setPublicAddress("10.26.14.210");

HazelcastInstance hazelcastInstance = Hazelcast.newHazelcastInstance(config);
```

This is how a cluster configuration using TCP/IP configuration looks like. The first step is to create a Config object and set the Multicast configuration to false. Then, you have to enable the TCP/IP configuration and add using the addMember method, introduce the IP address of the other nodes. Finally, you have to specify which interface you want the computer running the code to use (in this case, the computer's IP is 10.26.14.210) by using the SetPublicAdress method. This way, the node can join the cluster using the IP address of the network interface that will be used to join the cluster. This is how the cluster will be deployed in this experiment.

The queues for both pusblishing the jobs and the results also needs to be created. This is how the queues will be created:

3.1.3 Data Structures

In first place we need to create the data structure that will be used to store the matrices. As told previously, an artificial serializable data structure will be used to store the matrices. This data structure will contain all the necessary information for both the manager node and the worker nodes to process the jobs: the position of the resulting element in the matrix C, the row of the first matrix and the column of the second matrix that need to be multiplied, and a field to store the result of the multiplication. This way, the manager node can easily divide the matrices in smaller matrices and send the jobs to the queue, and the worker nodes can easily process the jobs and store the results in the same object before sending it to the results queue. This is how the data structure looks like:

```java
import java.io.Serializable;

public class RowMultiplicationTask implements Serializable {
    private final int row;
    private final int col;
    private final double[] rowData;
    private final double[] colData;
    private double result;

    public RowMultiplicationTask(int row, int col, double[] rowData, double[] colData)
     {
        this.row = row;
        this.col = col;
        this.rowData = rowData;
        this.colData = colData;
        this.result = 0;
    }

    public int getRow() { return row; }
    public int getCol() { return col; }
    public double[] getRowData() { return rowData; }
    public double[] getColData() { return colData; }
    public RowMultiplicationTask setResult(double result) {
        this.result = result;
        return this;
    }
    public double getResult() { return result; }
}
```

The only method that is not self-explanatory is the setResult method. This method is used to store the result of the multiplication in the object. This way, the worker nodes can easily store the results in the object before sending it to the results queue. This is how the data structure will be used to store the matrices. The setResut method returns the object itself, so it can be used in a fluent way. This way, the worker nodes can easily store the results in the object before sending it to the results queue. Please note that the data structure is serializable, so it can be sent to the queue and sent through the network.

3.1.4 Manager Node

Is the one responsible for dividing the matrices in smaller matrices and sending the jobs to the queue. Once all the tasks have been sent to the queue, the manager will also help the worker nodes to process the jobs. When there are no jobs left on the job queue, it starts polling messages from the results queue and starts storing the results in the matrix C. The algorithm of the manager is the following:

Algorithm 2 Manager Behaviour

```
1: Initialize matrices $a$, $b$, and $c$
2: Divide the matrices and send tasks to $jobQueue$
3: while $resultsQueue$ is not empty or there are pending tasks do
4:     Poll results from $resultsQueue$ and store them in matrix $c$
5: end while
6: Poll results from $resultsQueue$ and store them in matrix $c$
7: Output: Matrix $c$
```

3.1.5 Worker Nodes

Their behaviour is even simpler than the manager node. They will be responsible for processing the jobs and storing the results in the results queue. The will wait endleslly for a job to be published in the jobs queue, and once a job is published, they will process the job and store the result in the results queue. This is how the worker nodes work:

Algorithm 3 Workers Behaviour

```
1: while $true$ do
2:     if $jobQueue$ is not empty then
3:         Poll jobs from $jobQueue$, perform the multiplication and send the result to $resultsQueue$
4:     else
5:         Keep polling until there is work to do
6:     end if
7: end while
```

3.1.6 Division of the Matrices

The division of the matrices is the most important part of the solution. The matrices will be divided in row * column multiplications. This way, the job consists in multiplying a row of the first matrix by a column of the second matrix. This way, the job can be easily divided among the different nodes. The manager node will be responsible for dividing the matrices in smaller matrices and sending the jobs to the queue. This is how the matrices will be divided:

Algorithm 4 Divide and Send Tasks

```
1: Input: Matrices $a$ and $b$, Queue $jobQueue$
2: $n \leftarrow$ size of matrix $a$ (assuming $a$ and $b$ are square matrices of size $n \times n$)
3: for $i = 0$ to $n - 1$ do
4:     for $j = 0$ to $n - 1$ do
5:         Extract row $i$ from matrix $a$ and assign it to $rowA$
6:         Initialize $colB$ as an array of size $n$
7:         for $k = 0$ to $n - 1$ do
8:             $colB[k] \leftarrow b[k][j]$
9:         end for
10:        Create a task to multiply row $i$ by column $j$
11:        Send the task to the queue $jobQueue$
12:    end for
13: end for
```

3.1.7 Multiplication of the rows and columns

Since the matrices have been divided in row * column multiplications, the multiplication of the matrices is very simple. The only thing that needs to be done is to multiply the row of the first matrix by the column of the second matrix and sum the results. This is how the multiplication of the matrices will be done (is an element-wise multiplication):

Algorithm 5 Multiply Row by Column

```
1: Input: Arrays $row$ and $col$
2: Initialize $mult \leftarrow 0$
3: for $k = 0$ to $row.length - 1$ do
4:     $mult \leftarrow mult + row[k] \times col[k]$
5: end for
6: Return: $mult$
```

3.1.8 Polling the results and storing them in the matrix C

Once all the jobs have been processed and the jobs queue is empty, the manager node will start polling the results from the results queue and storing them in the matrix C. This is how the manager node will store the results in the matrix C:

Algorithm 6 Polling and Storing Results

```
1: while true do
2:     $task \leftarrow resultQueue.poll()$
3:     if $task \neq null$ then
4:         $c[task.getRow()][task.getCol()] \leftarrow task.getResult()$
5:     else
6:         break
7:     end if
8: end while
```

3.2 Frequent Item Set Problem

The solution to the frequent item set problem will be implemented using MapReduce and the mrjob library. Our case study was presented in the problem statement: given a list of baskets, find the most frequent items that are bought together. In the methodology it will be explained both the map and the reduce functions. The map function will be responsible for processing the data set and emitting key-value pairs. The reduce function will be responsible for processing the key-value pairs and emitting the final result.

3.2.1 Map Function

```python
def mapper(self, _, line):
    items = line.strip().split()
    for r in range(2, len(items) + 1):
        for comb in combinations(items, r):
            yield tuple(sorted(comb)), 1
```

This is how the map function looks like. The map function is responsible for processing the data set and emitting key-value pairs. The map function receives a line of the data set and splits it into items. Then, it generates all the possible combinations of items and emits a key-value pair for each combination. The key is the combination of items and the value is 1. This way, the map function emits a key-value pair for each combination of items.

3.2.2 Reduce Function

```python
def reducer(self, pair, counts):
    total_count = sum(counts)
    if total_count >= 2:
        yield pair, total_count
```

The reducer is very simple, it just sums the counts of each pair and emits the final result. If the total count is greater than or equal to 2, it emits the pair and the total count. This way, the reduce function processes the key-value pairs and emits the final result. Needless to say that the threshold can be changed by changing the value in the if statement to the desired value.

3.3 Software used

Intellij Idea along with a maven project were chosen for running and editing Java code, and then Hazelcast for the cluster deployment - added as a dependency in the pom.xml file. Visual Studio for writing Python code and MRJob as the MapReduce framework:

```
- Java 17
- Hazelcast 5.4.0
- Python 3.9
- MRJob
```

4 Experiments

As stated in the introduction, no experiments can be perfomed against the matrix multiplication problem nor the frequent item set problem. Therefore, the performance of the solution will be discussed theoretically.

Regarding the matrix multiplication solution with hazelcast, I can say that the solution itself works as expected: the manager is able to divide the matrix and push the jobs into the queue and then retrieve the results, the workers perform their task perfectly... All the parts of the solution work as expected. In terms of performance, we should expect this solution to be better for large matrices so that the overhead of both the process of dividing the matrices and the communication between the nodes is compensated by the parallelization of the task. I can bet that for matrices larger than 1024x1024 more or less, the solution will be faster than the sequential solution, and even some of the parallel solutions or better than the strassen algorithm, for example. However, for smaller matrices, the overhead of the process of dividing the matrices and the communication between the nodes will make the solution slower than the sequential solution. This is the main drawback of the solution: the overhead of the process of dividing the matrices and the communication between the nodes. However, this is a problem that can be solved by using a more powerful computer or by using a cluster of computers. In this way, the solution can be scaled to process larger matrices in a faster way.

This application is highly scalable since multiple nodes can be added as workers, increasing the number of jobs that can be processed in parallel, but not only that, the manager node can also be scaled, so the division of the matrices can be done in parallel.

Imagine the following situation: we want to multiply a 1million x 1million matrix by a 1million x 1million matrix and we have prepared a cluster of 1000 nodes. There will be much more nodes than jobs to be processed, so the manager node will be the bottleneck of the system. In this case, we can add more manager nodes to divide the matrices in parallel. This way, the solution can be scaled to process larger matrices in a faster way. This is the main advantage of the solution: it can be scaled to process larger matrices in a faster way. We can therefore not only increase the number of worker nodes, but also the number of manager nodes.

We may encounter some issues along the way regarding heap memory as well: when you run an application, the JVM allocates a certain amount of memory to the application. This memory is called the heap memory. The heap memory is used to store the objects that are created by the application. If the application creates too many objects and the heap memory is full, the JVM will throw an OutOfMemoryError. This is a common problem in Java applications, and it can be solved by increasing the heap memory. This problem will occur if it is tried to multiply a very big matrix and there is not enough workers: heap memory will be full and the JVM will throw an OutOfMemoryError. This is a problem that can be solved by increasing the heap memory or by adding more worker nodes. This is the main drawback of the solution: the heap memory can be full if there are not enough worker nodes.

Another problem that may be faced is that, when trying to deploy the cluster, if they are running inside a docker container, the nodes may not be able to communicate with each other. There is not a solution for this that will work 100% of the time, but one of the most common solutions is to use the host network, if the container is running in a linux machine. Otherwise, the best shot is to use the setPublicAddress method of the TcpIpConfig object, as stated previously.

5 Conclusions

In previous stages we came to the conclusion that even though these tasks (increasing the performance of the algorithm, trying to multiply matrix in a parallel way...) were in some way naïve, with few adjustemnts, all those methods and approaches could be introduced in real life situations. However, in this stage, the solution is not naïve at all, and it is a real life solution. The solution is a distributed solution that can be used to process large matrices in a faster way, or any other task that can be divided in smaller tasks.

I have also learned a lot of things about distributed computing. I have learned how to deploy a cluster of computers, how to use different data structures to store the data set, how to divide the data set in smaller data sets, how to process the data set in a distributed way, how to scale the solution to process larger data sets in a faster way... I have learned a lot of things about distributed computing, and I have learned how to apply those concepts to solve real life problems.

This stage in particular made me realize that sometimes developers use libraries, frameworks, or tools without knowing how they work, treating them as a black box or as if they were magic. This is a mistake, since it is very important to know how the tools that we are using work, so we can use them in the most efficient way. The thing is that sometimes, working with hazelcast seems like magic, since it is a distributed computing platform that allows to deploy a cluster of computers, so they can communicate and share the data in almost a magical way. And it is very important to realize that hazelcast is not magic and all data structures that you want to share with the other nodes have to be serializable so they can be sent through the network, that memory is not infinite and that the heap memory can be full if there are not enough worker nodes, that the nodes may not be able to communicate with each other if they are running inside a docker container... It is therefore very important to know not just how hazelcast works but all the libraries and frameworks we use so that we use them properly.

In fact, with the solution of the MapReduce problem, we got to see how to use a distributed framework to solve a problem that is used in data mining. And this is the main conclusion of the project: distributed computing can be used for a wide range of problems, not only for matrix multiplication. Distributed solutions can be applied to almost any problem that can be divided in smaller tasks. This is the main conclusion of the project.

All these stages have been a great experience for me, and I have learned a lot of things along the way. The first one is, of course, how to benchmark a solution in a proper way, and not just by using timestamps inside the code. The second one is the ability to search different solutions for the same problem, and the ability to compare them. The third one is the ability to think out of the box and use different point of views in order to come up for the best solution (choosing the fastest programming language for each situation, trying to improve the algorithm itself, trying to add parallelization to the code, seeing if the problem can be solved in a distributed way...).

6 Future Work

The next step could be test the matrix multiplication application in a real environment, i.e, got to the lab at the university and test it. I wanted to do that this week, but I was very busy with the group assignment and I could not do it. It would have been very interesting to see how the solution works in a real environment, and to see how it scales to process larger matrices in a faster way.

Another interesting thing to do would be to try to solve the frequent item set problem using hazelcast. This way, we could compare the performance of the solution with the solution that was implemented using MapReduce. This would be a very interesting experiment, since we could see how the two solutions compare in terms of performance.

The same could be done with the matrix multiplication problem: we could compare the performance of the solution that was implemented using hazelcast with the performance of the solution that was implemented using MapReduce.